Introduzione agli Algoritmi
con spunti per la soluzione

12 Giugno 2024

Esercizio 1 (15 punti):
Si consideri la seguente funzione:

```
def es1(n):
    if n <= 2: return 1
    m = 1
    s = n
    while m*m*m < n:
        m += 1
        s += 5
    b = 2*es1(m) + s*es1(m)
    for i in range(m+1):
        b -= 2
    return es1(i) + b - s
```

a) Si imposti la relazione di ricorrenza che ne definisce il tempo di esecuzione giustificando dettagliatamente l'equazione ottenuta.

b) Si risolva la ricorrenza usando uno dei metodi a scelta, dettagliando i passaggi del calcolo e giustificando ogni affermazione.

Ad ogni iterazione del $while$ **la variabile** m **si incrementa di** 1 **il suo valore, edopo** $m^{\frac{1}{3}}$ **iterazioni si avrà** $m * m * m = n$ **e si**

uscirà dal $while$ **. Al termine del** $while$ **si ha quindi** $m = n^{\frac{1}{3}}$ **e il** for **verrà di conseguenza iterato** $n^{\frac{1}{3}}$ **volte. Avvengono poi in totale** 3 **chiamate ricorsive ad un sottoproblema di dimensione** $m = n^{\frac{1}{3}}$ **Per quanto detto la relazione di ricorrenza che caratterizza il tempo della funzione è:**

- $T(n) = 3T(\sqrt[3]{n}) + \Theta(\sqrt[3]{n})$
- $T(2) = \Theta(1)$

Il modo più semplice per risolvere questa ricorrenza è ricorrere al metodo di sostituzione per dimostrare così che $T(n) = \Theta(\sqrt[3]{n})$**.**
Poiché il metodo più usato dagli studenti è stato quello iterativo riporto di seguito i passaggi di quest'ultima strada:

$$\begin{aligned} T(n) &= 3T\left(n^{\frac{1}{3}}\right) + \Theta\left(n^{\frac{1}{3}}\right) \\ &= 3\left(3T\left(n^{\frac{1}{3^2}}\right) + \Theta\left(n^{\frac{1}{3^2}}\right)\right) + \Theta\left(n^{\frac{1}{3}}\right) \\ &= 3^2 T\left(n^{\frac{1}{3^2}}\right) + 3 \cdot \Theta\left(n^{\frac{1}{3^2}}\right) + \Theta\left(n^{\frac{1}{3}}\right) \\ &= \ldots\ldots \\ &= 3^i T\left(n^{\frac{1}{3^i}}\right) + \sum_{j=1}^{i} 3^{j-1} \cdot \Theta\left(n^{\frac{1}{3^j}}\right) \end{aligned}$$

l'iterazione termina quando $n^{\frac{1}{3^i}} = 2$ **vale a dire quando** $\frac{1}{3^i}\log_2 n = 1$ **e quindi** $\log_2 n = 3^i$ **e** $i = \log_3 \log_2 n$**. A quel punto otteniamo:**

$$\begin{aligned} T(n) &= 3^{\log_3 \log_2 n} T(2) + \sum_{j=1}^{\log_3 \log_2 n} 3^{j-1} \cdot \Theta\left(n^{\frac{1}{3^j}}\right) \\ &= \log_2 n \Theta(1) + \sum_{j=1}^{i-1} 3^{j-1} \cdot \Theta\left(n^{\frac{1}{3^j}}\right) \end{aligned}$$

Per la somma si ha

$$\sum_{j=1}^{\log_3 \log_2 n} 3^{j-1} \cdot \Theta\left(n^{\frac{1}{3^j}}\right) = \Theta\left(n^{\frac{1}{3}}\right)$$

(basti pensare che i termini della somma sono decrescenti e il termine dominante si ha per $j = 1$ per il quale vale $3^{j-1} \cdot \Theta\left(n^{\frac{1}{3^j}}\right) = \Theta(n^{\frac{1}{3}})$).
La complessità della funzione è dunque:

$$\Theta(\log n) + \Theta\left(n^{\frac{1}{3}}\right) = \Theta\left(n^{\frac{1}{3}}\right).$$

Esercizio 2 (10 punti): Si definisce punto di sella di una matrice quell'elemento che gode della proprietà di essere simultaneamente minimo di riga e massimo di colonna. Si progetti un algoritmo che, data una matrice quadrata M di interi distinti restituisce una coppia di coordinate $(i,\ j)$ in cui è presente un punto di sella di M, restituisce $None$ se la matrice non ha punti di sella.
L'algoritmo deve avere complessità $\Theta(n^2)$.
Dell'algoritmo proposto:

a) si scriva lo pseudocodice opportunamente commentato;

b) si giustifichi il costo computazionale.

a) Possiamo applicare al definizione stessa di punto di sella: per ogni riga i calcoliamo la posizione j in cui compare l'elemento minimo e poi calcoliamo la posizione del massimo della colonna j, se questa posizione è i allora abbiamo trovato il punto di sella e restituiamo le coordinate $(i,\ j)$, in caso contrario passiamo alla riga successiva. Se in nessuna delle righe troviamo il punto di sella terminiamo restituendo $None$.

Segue un possibile codice Python dell'idea proposta. Dove la funzione $minR(i, M)$ restituisce la posizione nella riga i di M in cui si trova il minimo, mentre $maxC(j, M)$ restituisce la posizione nella colonna j di M in cui si trova il massimo.

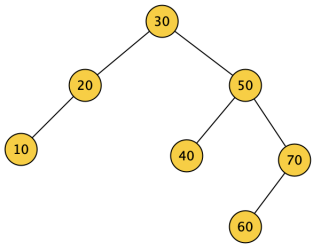
```python
def es2(M):
    n= len(M)
    for i in range(n):
        j= minR(i,M)
        if maxC(j,M)==i:
            return i,j


def minR(i,M):
    n, mj = len(M), 0
    for j in range(1,n):
        if M[i][j]<M[i][mj]: mj = j
    return mj


def maxC(j, M):
    n, mi = len(M), 0
    for i in range(1,n):
        if M[i][j]>M[mi][j]: mi=i
    return mi
```

b) Il costo di una chiamata a $minR$ o a $maxC$ è $\Theta(n)$ quindi ogni iterazione del for nel programma principale richiede $\Theta(n)$ poiché le iterazioni sono n vla complessità dell'algoritmo risulterà $\Theta(n^2)$.

Esercizio 3 (10 punti): Dato un albero binario di ricerca ed un intero k si vuole individuare la k-esima chiave dell'albero se queste fossero messe in ordine. Ad esempio per l'albero in figura:



- con $k = 1$ la risposta è 10,
- per $k = 5$ la risposta è 50
- per $k = 9$ la risposta è $None$ perché l'albero ha meno di k nodi.

L'albero è memorizzato tramite puntatori e record di quattro campi: il campo key contenente il valore, i campi $left$ e $right$ con i puntatori al figlio sinistro e al figlio destro, rispettivamente (questi puntatori valgono $None$ in mancanza del figlio), ed il campo num con l'indicazione del numero dei nodi nel sottoalbero in esso radicato.

Si progetti un algoritmo RICORSIVO che dato il puntatore alla radice dell'albero e l'intero k, risolva il problema in un tempo computazionale $O(h)$ dove h è l'altezza dell'albero.
Dell'algoritmo proposto:

a) si scriva lo pseudocodice opportunamente commentato;

b) si giustifichi il costo computazionale.

NOTA BENE: nello pseudocodice dell'algoritmo ricorsivo **non** si deve far uso di variabili globali.

a) **Per ciascun nodo p, grazie al campo num suo e dei suoi figli (in realtà del solo figlio sinistro), p è in grado di scoprire se la chiave da stampare sia la sua, si trovi nel sottoalbero di sinistra, o nel sottoalbero di destra. Il primo caso si ha o quando p non ha figlio sinistro e $k = 1$, oppure quando il figlio sinistro esiste e nel sottoalbero sinistro ci sono esattamente $k - 1$ nodi. Gli altri due casi sono gestiti ricorsivamente. Da notare che, mentre nella chiamata ricorsiva a sinistra k rimane inalterato, nella chiamata a destra bisogna tenere conto del fatto che, andando verso destra, alcune chiavi sono state di fatto già conteggiate e quindi il valore di k nella chiamata cambia.**

   Più sotto viene riportata una possibile codifica in Python dell'algoritmo descritto che non utilizza variabili globali:

b) **Ad ogni chiamata ricorsiva si discende di un livello nell'albero di altezza h di conseguenza per i tempi di calcolo abbiamo la seguente equazione di ricorrenza :**

```python
def cerca(p,k):
    if p==None or k>p.num:
        return None
    if p.left == None:
        if k==1:
            return p.key
        else:
            return cerca(p.right, k-1)
    if p.left.num >= k:
        return cerca(p.left, k)
    if p.left.num == k-1:
        return p.key
    return cerca(p.right, k-1-p.left.num)
```

- $T(h) \leq T(h-1) + \Theta(1)$
- $T(0) = \Theta(1)$

L'equazione si può risolvere con il metodo iterativo (che va esplicitamente svolto) dando come soluzione $O(h)$. Ogni passaggio va dettagliato.